Red–Black Trees

Presented By

Shivam Verma - A125020
Somyajit Jitendra S. - A125022

Department of CSE, IIIT Bhubaneswar
November 2025

Outline

Background

Binary Search Trees are widely used due to their efficient searching ability. However, their performance depends entirely on the height of the tree.

- Balanced trees give fast performance.
- Unbalanced trees degrade to linear structures.
- Red–Black Trees guarantee balance automatically.

Why Red–Black Trees?

Real-world applications require guaranteed performance.

- Databases require predictable access time.
- Operating systems cannot afford worst-case slowdowns.
- Red–Black Trees provide guaranteed $O(\log n)$ time.

BST Characteristics

A Binary Search Tree maintains an ordered structure.

- Left subtree contains smaller keys.
- Right subtree contains larger keys.
- Efficient search when tree is balanced.

However, BSTs do not enforce balance.

Skewed Tree Formation

If elements are inserted in sorted order, BST becomes skewed.

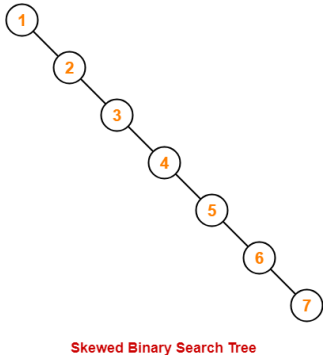


Figure: Red–Black Tree Structure

Search time becomes $O(n)$.

Key Insight

To maintain efficiency, tree height must be controlled.

- Height directly impacts performance.
- Self-balancing trees restructure automatically.
- Balance is preserved after insertions and deletions.

Self-Balancing Trees

Common self-balancing tree structures include:

- AVL Trees – strictly balanced
- Red–Black Trees – loosely balanced
- B-Trees – disk-optimized

Red–Black Trees require fewer rotations than AVL Trees.

Definition and Core Idea

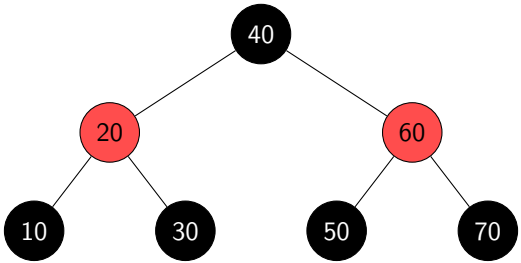
- A Red–Black Tree is a self-balancing Binary Search Tree that maintains height automatically.
- It follows all standard BST ordering rules for storing and searching keys.
- Each node contains an additional color attribute, either red or black.
- The color information is used to impose structural constraints on the tree.
- These constraints ensure that the tree never becomes highly skewed.

How Coloring Maintains Balance

- Coloring rules restrict how red and black nodes can appear along any root-to-leaf path.
- Black nodes act as anchors that regulate the overall height of the tree.
- Red nodes allow limited imbalance without immediately violating tree properties.
- The coloring strategy guarantees that no path is more than twice as long as another.
- This approach achieves balance with fewer rotations compared to strictly balanced trees.

Example of a Red–Black Tree

Balanced Structure with Coloring



- The root node is black, satisfying the root property.
- Red nodes have only black children, preventing adjacent red nodes.
- Every root-to-leaf path contains the same number of black nodes.
- The tree remains balanced while preserving BST ordering.

Fundamental Rules

- Every node in the tree is assigned exactly one color, either red or black.
- The root node of the tree is always colored black to maintain uniform structure.
- All leaf nodes (NIL or null children) are considered black by definition.
- A red node cannot have a red parent or red child, preventing consecutive red nodes.
- Every path from a node to its descendant NIL leaves contains the same number of black nodes.

Balancing Principle

- Black height is the number of black nodes on any path from a node to a NIL leaf.
- All root-to-leaf paths in a Red–Black Tree have the same black height.
- This constraint prevents the tree from becoming highly unbalanced.
- Black nodes provide the structural balance of the tree.

Why Height Remains Logarithmic

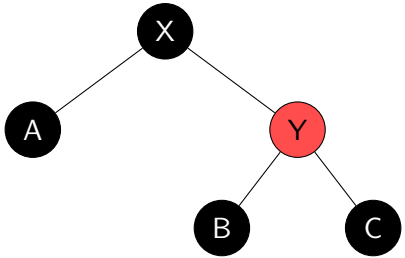
$$h \leq 2 \log_2(n+1)$$

- The black-height property ensures a minimum number of black nodes on every path.
- Red nodes cannot appear consecutively, limiting how tall any path can grow.
- Every longest path contains alternating red and black nodes at most.
- As a result, the height of a Red–Black Tree is bounded logarithmically.

Purpose of Rotations

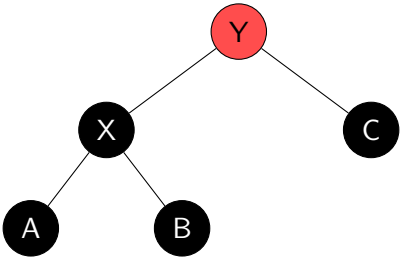
- Tree rotations are local restructuring operations used to restore balance.
- Rotations modify parent–child links without changing BST key order.
- Only a small portion of the tree is affected during a rotation.
- Rotations are applied when Red–Black Tree properties are violated.
- Both insertion and deletion algorithms rely on rotations.

Before Rotation (Right-Heavy Structure)



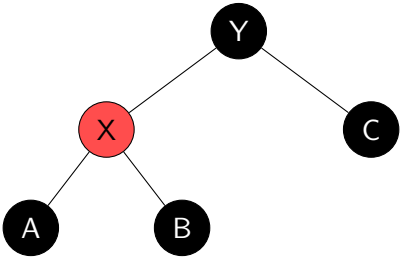
- Node X has a heavier right subtree rooted at Y.
- The imbalance violates height constraints after insertion.
- A left rotation is required at node X.

After Rotation



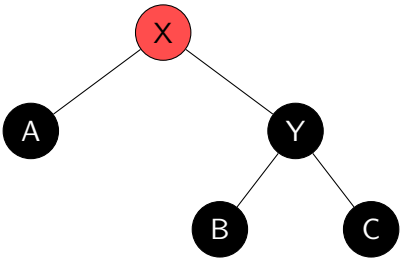
- Node Y moves up and becomes the new subtree root.
- Node X becomes the left child of Y.
- Subtree B is repositioned to preserve BST ordering.

Before Rotation (Left-Heavy Structure)



- Node Y has a heavier left subtree rooted at X.
- The imbalance occurs due to recent insertion or deletion.
- A right rotation is required at node Y.

After Rotation



- Node X moves up to become the new subtree root.
- Node Y becomes the right child of X.
- Subtree B is repositioned without violating BST rules.

High-Level Procedure

- Insertion starts by placing the new key using standard BST insertion rules.
- The newly inserted node is always colored red initially.
- If the parent node is black, no property is violated.
- If the parent node is red, Red–Black Tree properties are violated.
- Violations are fixed using recoloring and rotations.

Why Insert the New Node as Red?

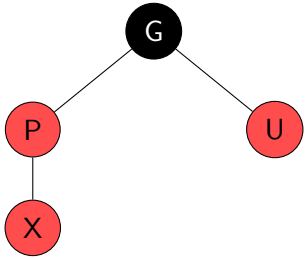
Design Choice

- Inserting a black node would increase the black height of some paths.
- A red node does not affect the black height of the tree.
- Initial red coloring makes it easier to restore balance.
- Recoloring and rotations are applied only if required.

Possible Violations

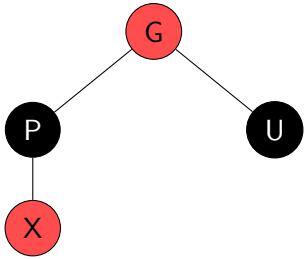
- Case 1: Parent and Uncle are both red.
- Case 2: Parent is red, Uncle is black, and nodes form a triangle.
- Case 3: Parent is red, Uncle is black, and nodes form a straight line.
- Each case applies specific recoloring and rotation rules.

Red Parent and Red Uncle



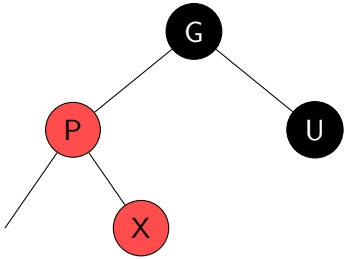
- The newly inserted node X has a red parent.
- The uncle node U is also red.
- This violates the rule that red nodes cannot have red children.

Recoloring to Restore Properties



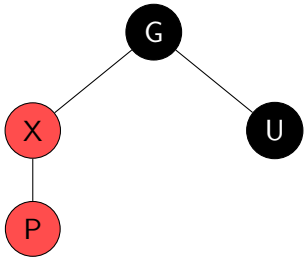
- Parent and uncle are recolored black.
- Grandparent is recolored red.
- The violation may propagate upward.

Triangle Configuration



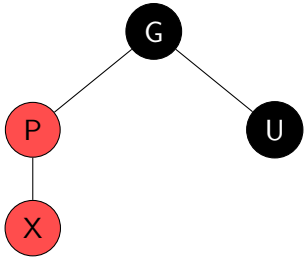
- The parent is red and the uncle is black.
- The newly inserted node forms a triangle shape.
- Direct rotation at the grandparent is not possible.

Rotation to Convert into Case 3



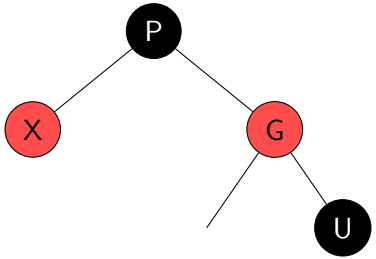
- A rotation is performed at the parent node.
- The structure is converted into a straight-line configuration.
- The problem now reduces to Case 3.

Straight-Line Configuration



- Parent is red and uncle is black.
- Nodes form a straight-line configuration.
- This configuration allows direct rotation at the grandparent.

Rotation and Recoloring



- A rotation is performed at the grandparent node.
- Colors of the parent and grandparent are swapped.
- All Red–Black Tree properties are restored.

Root Recoloring Rule

- After all insertion fix-up cases are applied, the root node is examined.
- If the root node is red, it is recolored black.
- This step ensures that the root property of Red–Black Trees is satisfied.
- Root recoloring does not affect black height consistency.
- With this step, all Red–Black Tree properties are fully restored.

Reasoning and Pseudocode

The logarithmic performance of Red–Black Trees is guaranteed by the height bound:

$$h \leq 2\log_2(n+1)$$

Search Operation (Pseudocode):

```
RB-SEARCH(node, key):
    if node == NIL or node.key == key:
        return node
    if key < node.key:
        return RB-SEARCH(node.left, key)
    else:
        return RB-SEARCH(node.right, key)
```

- Each recursive call moves one level down the tree.
- Maximum recursion depth equals the height of the tree.
- Since height is $O(\log n)$, the operation is logarithmic.

Insertion Operation

- Red–Black Tree insertion begins with a standard Binary Search Tree insertion.
- The BST insertion step takes $O(h)$ time, where h is the height of the tree.
- The newly inserted node is colored red, which does not affect black height.
- The fix-up procedure uses recoloring and at most two rotations.
- Each fix-up step moves upward toward the root and runs at most $O(h)$ times.
- Since the tree height is bounded by $O(\log n)$, insertion runs in $O(\log n)$ time.

Deletion Operation

- Deletion starts with a standard BST deletion, which takes $O(h)$ time.
- If the deleted node is red, no Red–Black Tree properties are violated.
- If a black node is deleted, the black-height property may be violated.
- The delete-fixup procedure restores balance using recoloring and rotations.
- Each fix-up iteration moves upward and executes at most $O(h)$ times.
- As the tree height is always $O(\log n)$, deletion runs in $O(\log n)$ time.

Best, Average, and Worst Case Analysis

The performance of Red–Black Trees depends on the height of the tree, which is always bounded logarithmically.

Operation	Best Case	Average Case	Worst Case
Search	$O(1)$	$O(\log n)$	$O(\log n)$
Insertion	$O(\log n)$	$O(\log n)$	$O(\log n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(\log n)$

- Best case for search occurs when the key is found at the root.
- Average case assumes random key distribution in the tree.
- Worst case remains logarithmic due to strict balancing rules.
- Unlike ordinary BSTs, Red–Black Trees never degrade to linear time.

Where Red–Black Trees Are Used

- Used in Java TreeMap and TreeSet for ordered data storage.
- Implemented in C++ STL map and set containers.
- Employed inside the Linux kernel for scheduling and memory management.
- Used in databases and file systems for indexing structures.

Why Red–Black Trees Are Preferred

- Guaranteed logarithmic performance for all dynamic operations.
- Requires fewer rotations compared to strictly balanced AVL Trees.
- Efficient for applications involving frequent insertions and deletions.
- Widely adopted and well-tested in real-world systems.

Practical Constraints

- Deletion algorithm is more complex compared to insertion.
- Slightly less balanced than AVL Trees in some scenarios.
- Requires additional memory for storing color information.

Key Takeaways

- Red–Black Trees are self-balancing Binary Search Trees using color constraints.
- Structural rules and black-height property keep the tree height logarithmic.
- Rotations and recoloring restore balance efficiently after insertions.
- All core operations run in $O(\log n)$ time in the worst case.

Final Remarks

- Red–Black Trees offer a strong balance between efficiency and simplicity.
- They are suitable for large, dynamic, and performance-critical datasets.
- Their practical design makes them a standard choice in system libraries.

References

1. T. H. Cormen et al., *Introduction to Algorithms*, MIT Press.
2. MIT OpenCourseWare — Red–Black Trees Lecture Notes.
3. Java Collections Framework Documentation.
4. Carnegie Mellon University Lecture Notes.
5. U T Austin Computer Science Lecture Notes.

Thank You